



Functional Programming

Week 10 – Input and Output, Connect Four

René Thiemann	Philipp Anrain	Marc Bußjäger	Benedikt Dornauer	Manuel Eberl
Christina Kohl	Sandra Reitingner	Christian Sternagel		

Department of Computer Science

Last Lecture

- **scoping** rules determine visibility of function names and variable names
- larger programs should be structured in **modules**
 - explicit **export-lists** to distinguish internal and external parts
 - import of modules instead of copying code
 - qualified imports and qualifiers are useful for resolving name conflicts
 - defaults
 - if program does not contain module declaration, `module Main where` is added
 - `import Prelude` is implicitly added, if no other imports of `Prelude` are present
- example

[illegible]

RT et al. (DCS @ UIBK)

Week 10

2/29

Input and Output in Haskell

I/O: Input and Output

- aim: communicate with the user
 - ask user for inputs
 - print answers
 - **outside** the GHCi read-eval-print-loop
 - stand-alone programs that neither require ghc-installation nor Haskell knowledge of user
- I/O is not restricted to text-based user-I/O
 - reading and writing of files
(e.g., compiler translates .hs to .exe, or .tex to .pdf)
 - reading and writing into memory
(mutable state, arrays)
 - reading and writing of network channels
(e.g., web-server and internet-browser)
 - start other programs and communicate with them
 - play/record sound, capture mouse-movements, ...

An Initial Example

- ```
main = do -- file: welcomeIO.hs
 putStrLn "Greetings! Please tell me your name."
 name <- getLine
 putStrLn $ "Welcome to Haskell's IO, " ++ name ++ "!"
```
- compile it with GHC (not GHCi) via  

```
$ ghc --make welcomeIO.hs
```
- and run it  

```
$./welcomeIO
```

 # welcomeIO.exe on Windows  
Greetings! Please tell me your name.  
Homer # this was typed in  
Welcome to Haskell's IO, Homer!
- notes
  - `putStrLn` – prints string followed by newline
  - `getLine` – reads line from standard input
  - new syntax: `do` and `<-`

## I/O and the Type System

- consider

```
ghci> :l welcomeIO.hs
ghci> :t putStrLn
putStrLn :: String -> IO ()
ghci> :t getLine
getLine :: IO String
ghci> :t main
main :: IO ()
```
- `IO a` is type of I/O actions delivering results of type `a` (in addition to their I/O operations)
- examples
  - `String -> IO ()` – after supplying a string, we obtain an I/O action (in case of `putStrLn`, “printing”)
  - `IO ()` – just perform I/O (in case of `main`, run our program)
  - `IO String` – do some I/O and deliver a string (in case of `getLine`, user-input)

## Combining I/O Actions

- I/O actions can be combined
- core building block: `bind` (syntax `>>=`)
  - (`>>=`) :: IO a -> (a -> IO b) -> IO b
- consider `act1 >>= \ x -> act2`
  - on evaluation, this expressions first performs action `act1`
  - the result of action `act1` is stored in `x`
  - afterwards action `act2` is performed (which may depend on `x`)
  - in total, both actions are performed and the result is that of `act2`
- ignoring results: (`>>`) :: IO a -> IO b -> IO b, `a1 >> a2 = a1 >>= \_ -> a2`
- example
 

```
putStrLn "Hi. What's your name?" >> -- ignore result, which is ()
getline >>= \ name -> -- store result in variable name
let answer = "Hello " ++ name in -- no I/O in this line
putStrLn answer -- final result from putStrLn: ()
```

  - the type of overall expression is `IO ()`, that of the last I/O action `putStrLn answer`
  - execution of actions is sequential, like in imperative programming

## Do-Notation

- there is special syntax for combinations of binds, lambdas and lets
 

```
do x <- act = act >>= \ x -> do block
 block

do act = act >> do block
 block

do let x = e = let x = e in do block
 block
```
- `putStrLn "Hi. What's your name?" >>
 getLine >>= \ name ->
 let answer = "Hello " ++ name in
 putStrLn answer`
 can be written as
 

```
do putStrLn "Hi. What's your name?"
 name <- getLine
 let answer = "Hello " ++ name -- no "in"!
 putStrLn answer
```
- as in `let`-syntax, `do`-blocks can also be written via `do {...; ...; ...}`

## Further Notes

- inside do-block, order is important; I/O actions are executed in order of appearance; result of block is result of **last** action
- `x <- a` is not available outside I/O actions, in particular there is no function of type `IO a -> a` which extracts the results of an action (of type `IO a`) without being an action itself (result type `a`)
  - once we are inside an IO action, we cannot escape
  - **strict separation between purely functional code and I/O**
  - when `IO a` does not appear inside type signature, we can be absolutely sure that no I/O ("side-effect") is performed
- `main :: IO ()` is the I/O action that is executed when running a compiled file via `ghc --make prog.hs` and then `./prog` (`prog.hs` must contain a module `Main` that exports `main`)

## Using Purely Functional Code Inside I/O Actions

```
-- reply is purely functional: no IO in type
reply :: String -> String
reply name =
 "Pleased to meet you, " ++ name ++ ".\n" ++
 "Your name contains " ++ n ++ " characters."
 where n = show $ length name

-- pure code can be invoked from I/O-part
main :: IO ()
main = do
 putStrLn "Greetings again. What's your name?"
 name <- getLine
 let niceReply = reply name
 putStrLn niceReply
```

- invoking purely functional code inside I/O is easy
- the other direction is not possible

## Some Predefined I/O Functions

- `return :: a -> IO a` – turn anything into an I/O action which does nothing
- `System.Environment.getArgs :: IO [String]` – get command line arguments
- `putChar :: Char -> IO ()` – print character
- `putStr :: String -> IO ()` – print string
- `putStrLn :: String -> IO ()` – print string followed by newline
- `getChar :: IO Char` – read single character from stdin
- `getLine :: IO String` – read line (no newline-character in result)
- `interact :: (String -> String) -> IO ()` – use function that gets input as string and produces output as string
- `type FilePath = String`
- `readFile :: FilePath -> IO String` – read file content
- `writeFile :: FilePath -> String -> IO ()`
- `appendFile :: FilePath -> String -> IO ()`

## Recursive I/O Actions

- branching and recursion is also possible with I/O actions
- example: implement `getLine` via `getChar`

```
import Prelude hiding (getLine)

getLine = do
 c <- getChar
 if c == '\n' -- branching
 then return ""
 else do
 l <- getLine -- recursion
 return $ c : l
```

## Examples – Imitating Some GNU Commands

- `cat.hs` – print file contents

```
import System.Environment (getArgs)
main = do
 [file] <- getArgs -- assume there is exactly one file
 s <- readFile file
 putStr s
```
- `wc.hs` – count number of lines/words/characters in input

```
count s = nl ++ " " ++ nw ++ " " ++ nc ++ "\n"
 where nl = show $ length $ lines s
 nw = show $ length $ words s
 nc = show $ length s
main = interact count
```
- `sort.hs` – sort input lines

```
import Data.List (sort)
main = interact (unlines . sort . lines)
```

## Laziness and I/O Actions

- consider a simple copying program

```
main = do -- imports omitted
 [src, dest] <- getArgs
 s <- readFile src
 writeFile dest s
 • readFile and writeFile are lazy, e.g., readFile only reads characters on demand
 • positive effect: large files can be copied without fully loading them into memory
```
- laziness might lead to problems

```
main = do -- imports omitted
 [file] <- getArgs
 s <- readFile file
 writeFile file (map toUpper s)
 • since readFile is lazy, when executing s <- readFile file nothing is read immediately
 • but then the same file should be opened for writing; conflict, which will result in error
 • solution: more fine-grained control via file-handles which explicitly open and close files, see lecture Operating Systems
```

## Higher-Order on I/O Actions

- `foreach` :: `[a] -> (a -> IO b) -> IO ()`

```
foreach [] io = return ()
foreach (a:as) io = do { io a; foreach as io }
```
- better `cat.hs`

```
main = do
 files <- getArgs
 if null files then interact id else do
 foreach files readAndPrint
 where readAndPrint file = do
 s <- readFile file
 putStr s
```

## Monads

- bind and do-notation are *not* fixed to I/O
- there exists a more general concept of *monads*
- example: also the *Maybe*-type is a monad

```
data Expr = Const Double | Div Expr Expr
eval :: Expr -> Maybe Double
eval (Const c) = return c
eval (Div expr1 expr2) = do
 x1 <- eval expr1
 x2 <- eval expr2
 if x2 == 0
 then Nothing
 else return (x1 / x2)
```
- monads won't be covered here, but they are the reason why the Haskell literature speaks about the *I/O-monad*

## Example Application: Connect Four

### Connect Four

- aim: implement **Connect Four**, MB Spiele



- with textual user interface

```
0123456
```

```
.....
.X0.X..
.X000X0
X0X0X0X
0XX0X00
XX0X00X
```

```
Player X to go
Choose one of [0,1,2,3,4,5,6]
```

### Connect Four: Implementation

- clear separation between
  - user interface (I/O)
    - ask for a move
    - print the current state
    - ...
  - game logic (purely functional code)
    - type to represent a state (board + next player)
    - perform a move
    - check for a winner
    - display a state as string
    - ...
- both parts would be written as two separate modules
  - `Logic` contains the game logic
  - `Main` contains the user interface and the `main` function

### Game Logic: Interface

- types: `State`, `Move` and `Player`
- constant `initState :: State`
- function `showPlayer :: Player -> String`
- function `showState :: State -> String`
- function `winningPlayer :: State -> Maybe Player`
- function `validMoves :: State -> [Move]`
- function `dropTile :: Move -> State -> State`
- in total
 

```
module Logic(State, Move, Player,
 initState, showPlayer, showState,
 winningPlayer, validMoves, dropTile) where
... -- details, which the user interface doesn't have to know
```

## The Read-Class

- class `Read` provides methods to convert `Strings` into other types
  - `read :: Read a => String -> a`
  - `readMaybe :: Read a => String -> Maybe a`  
import of module `Text.Read` required
  - when using `read`, often the type `a` has to be chosen explicitly
  - examples
    - `(read "(41, True)" :: (Integer, Bool)) = (41, True)`
    - `(read "(41, True)" :: (Integer, Integer)) = error ...`
    - `(readMaybe "1" :: Maybe Integer) = Just 1`
    - `(readMaybe "one" :: Maybe Integer) = Nothing`
- for the `Logic` module, we assume that the type `Move` is an instance of `Show` and `Read`

## User Interface

```
module Main(main) where -- module name must be "Main" for compilation
import Logic

main = do
 putStrLn "Welcome to Connect Four"
 game initState

game state = do
 putStrLn $ showState state
 case winningPlayer state of
 Just player -> putStrLn $ showPlayer player ++ " wins!"
 Nothing -> let moves = validMoves state in
 if null moves then putStrLn "Game ends in draw."
 else do
 putStrLn $ "Choose one of " ++ show moves ++ ": "
 moveStr <- getLine
 let move = (read moveStr :: Move)
 game (dropTile move state)
```

## Game Logic: Encoding a State and Initial State

```
type Tile = Int -- 0, 1, or 2
type Player = Int -- 1 and 2
type Move = Int -- column number
data State = State Player [[Tile]] -- list of rows

empty :: Tile
empty = 0

numRows, numCols :: Int
numRows = 6
numCols = 7

startPlayer :: Player
startPlayer = 1

initState :: State
initState = State startPlayer
 (replicate numRows (replicate numCols empty))
```

## Game Logic: Valid Moves and Displaying a State

```
validMoves :: State -> [Move]
validMoves (State _ rows) =
 map fst . filter ((== empty) . snd) . zip [0 .. numCols - 1] $ head rows

showPlayer :: Player -> String
showPlayer 1 = "X"
showPlayer 2 = "O"

showTile :: Tile -> Char
showTile t = if t == empty then '.' else head $ showPlayer t

showState :: State -> String
showState (State player rows) = unlines $
 map (head . show) [0 .. numCols - 1] :
 map (map showTile) rows
 ++ ["\nPlayer " ++ showPlayer player ++ " to go"]
```

## Game Logic: Making a Move

```
otherPlayer :: Player -> Player
otherPlayer = (3 -)
```

```
dropTile :: Move -> State -> State
dropTile col (State player rows) = State
 (otherPlayer player)
 (reverse $ dropAux $ reverse rows)
 where
 dropAux (row : rows) =
 case splitAt col row of
 (first, t : last) ->
 if t == empty
 then (first ++ player : last) : rows
 else row : dropAux rows
```

## Game Logic: Winning Player

```
winningRow :: Player -> [Tile] -> Bool
winningRow player [] = False
winningRow player row = take 4 row == replicate 4 player
 || winningRow player (tail row)
```

```
transpose ([] : _) = []
transpose xs = map head xs : transpose (map tail xs)
```

```
winningPlayer :: State -> Maybe Player
winningPlayer (State player rows) =
 let prevPlayer = otherPlayer player
 longRows = rows ++ transpose rows -- ++ diags rows
 in if any (winningRow prevPlayer) longRows
 then Just prevPlayer
 else Nothing
```

## Connect Four: Final Remarks

- implementation is quite basic
  - diagonal winning-condition missing
  - crashes when invalid moves are entered
  - no iterated matches
- exercise: improve implementation

## Summary

## Summary

- in Haskell I/O is possible, `IO a` is type of I/O-actions with result of type `a`
- clear separation between purely functional and I/O-code
- multiple actions can be connected via `(>>=)` or `do`-blocks
- several predefined functions to access I/O
- more information on I/O in Haskell:  
<http://book.realworldhaskell.org/read/io.html>
- `Read` class provides method `read :: String -> a`, opposite to `Show`
- connect four: separate implementation of game logic (pure) and user interface (I/O)